Escuela
Politécnica
Superior

Generación de logotipos mediante redes generativas antagónicas

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Autor:
Enrique Mas Candela
Tutor:
Jorge Calvo Zaragoza

Universitat d'Alacant
Universidad de Alicante

Septiembre 2020

Generación de logotipos mediante redes generativas antagónicas

Síntesis de imagen a partir de bocetos con técnicas de Deep Learning

Autor
Enrique Mas Candela

Tutor
Jorge Calvo Zaragoza
Departamento de Lenguajes y Sistemas Informáticos (DLSI)

Máster Universitario en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Septiembre 2020

Preámbulo

Este Trabajo de Fin de Máster tiene como objetivo desarrollar un sistema informático basado en Inteligencia Artificial que sea capaz de generar logotipos realistas a partir de simples bocetos. Para ello, se utilizarán técnicas modernas basadas en redes neuronales, en concreto redes generativas antagónicas (*Generative Adversarial Networks*, GAN). Esta memoria recoge el desarrollo, implementación y resultados del proyecto, así como los fundamentos y el estado actual de redes neuronales de tipo GAN.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

La creatividad siempre ha sido una cualidad atribuida a las personas, mientras que las máquinas tradicionalmente se han encargado de realizar tareas más mecánicas y programables. No obstante, con la llegada de la computación y el auge de la Inteligencia Artificial, este escenario ha cambiado drásticamente. Actualmente, la tendencia en este contexto la marcan las redes neuronales, entre las cuales se pueden destacar las redes generativas antagónicas (*Generative Adversarial Networks*, GAN). Gracias a esta tecnología, se ha demostrado que un ordenador también puede ser creativo, dando una vuelta de tuerca a las capacidades que los modelos generativos habían tenido hasta la fecha.

Este tipo de redes prometen acabar con la concepción clásica, mencionada anteriormente, de que la creatividad es solo cosa de humanos. A día de hoy ya podemos ver ejemplos de obras de arte generadas por una red neuronal que son difícilmente diferenciables de las obras de un humano. Esto nos permite realizarnos algunas preguntas: ¿podrá un algoritmo llegar al nivel de creatividad de un humano? ¿conseguirán las máquinas suplantar a los humanos a la hora de crear contenido original? Pese que a día de hoy estamos lejos de responder con certeza estas preguntas, sí que sabemos que, gracias a las GAN, los modelos generativos han llegado a un punto que nunca nadie imaginaría pese que aun no se ha alcanzado el límite al potencial que éstas tienen. Las GAN han tenido atenta a la comunidad científica desde que se presentaron y es que la base de su funcionamiento es tan disruptiva y simple a la vez que a nadie se le había ocurrido antes. Desde entonces no han parado de surgir ideas de aplicaciones de estas redes con increíbles resultados, no solo en generación de imágenes sino también de contenido como texto o música.

En este proyecto se pretende desarrolla un sistema computacional basado en GAN que sea capaz de generar logotipos de forma automática, a partir de bocetos hechos a mano que sirvan como base de la síntesis. Cabe destacar que, aunque los resultados finales obtenidos en el proyecto tienen apariencia de un logotipo realista, veremos que la generación todavía tiene ruido o imperfecciones, así como cierta falta de definición y detalle. Es por ello que no se pretende que el logotipo generado quede listo para ser usado directamente, por lo que de ningún modo se reemplaza completamente la tarea de un diseñador. Por otra parte, el sistema se concibe como punto de partida a la creación de logotipos o imágenes corporativas dando pie a una gran fuente de inspiración para realizar estos logotipos.

Aunque en este proyecto, el enfoque es eminentemente técnico, la solución puede dar pie a un producto software con todo tipo de potenciales aplicaciones que podrían cambiar la forma en la que a día de hoy se crea un logotipo. Por ejemplo, esta aplicación podría colaborar en la inspiración de un artista a la hora de diseñar un nuevo logotipo o ayudar a una empresa a decidir cómo enfocar su imagen corporativa generando decenas de logotipos, con la misma base (boceto), y estilos muy diferentes. Como producto, además, podría ofrecerse tanto de manera independiente como ser integrado en software de diseño gráfico existente, funcionando como herramienta adicional.

1.1 Motivación

La motivación principal de este proyecto es poder profundizar en el uso de técnicas vanguardistas de Inteligencia Artificial basadas en redes neuronales artificiales e introducirme en el mundo de las GAN. Tras un estudio preliminar sobre este tipo de tecnologías, pretendo llevar a la práctica los conocimientos adquiridos entrenando un sistema capaz de *crear* contenido por sí mismo. El aprendizaje sobre redes neuronales artificiales supone, personalmente, objeto de gran inquietud en los últimos años. Es por ello que se ha utilizado este proyecto para, además de aprender, realizar una implementación propia de un sistema capaz de resolver un problema real.

1.2 Objetivos

Como se ha comentado anteriormente, el principal objetivo de este proyecto es el desarrollo de un sistema basado en GAN que, de manera automática, reciba bocetos hechos a mano y los convierta en logotipos realistas. Para llevar a cabo el proyecto se han definido los siguientes objetivos específicos:

- Análisis del estado de la cuestión sobre las GAN.
- Estudiar problemas similares y la forma de solucionarlos.
- Selección de una base de datos apropiada para realizar el entrenamiento de la GAN.
- Diseñar e implementar distintos tipos de GAN, capaces de resolver el problema planteado.
- Experimentación con distintas arquitecturas y análisis de sus resultados..

Por otra parte, a nivel personal, el objetivo del proyecto es aprender técnicas vanguardistas de redes neuronales profundas, con las cuales atacar problemas abiertos e implementar modelos novedosos que puedan llegar a integrarse en productos existentes o incluso conformar uno por sí mismo.

2 Marco teórico

La síntesis de imágenes a partir de bocetos se realizará mediante GAN, las cuales se exponen en el siguiente capítulo. Además, se introducen los conceptos del aprendizaje automático y redes neuronales artificiales, paradigimas sobre las cuales se construyen las GAN.

2.1 Aprendizaje automático

El aprendizaje automático (*Machine Learning*, ML) es una rama de la Inteligencia Artificial que tiene como objetivo desarrollar técnicas que permitan a los ordenadores aprendan a resolver problemas de manera autónoma, sin ser programados explícitamente. Este aprendizaje se realiza usando datos a partir de los cuales estas técnicas tratan de aprender la tarea.

Hay distintas maneras de clasificar las técnicas de ML. Una de ellas es en función de la naturaleza de los datos. Según este criterio tenemos 3 tipos distintos:

- **Aprendizaje supervisado**. El aprendizaje supervisado es un tipo de ML que produce una salida a partir de una entrada en base a ejemplos de entrada-salida. Se entrena a partir de un conjunto de datos **etiquetado**, del que para cada elemento del conjunto se dispone de una salida esperada, que es la que el algoritmo pretende inferir a partir de los datos de entrada. Podemos utilizar el aprendizaje supervisado, por ejemplo, en tareas de clasificación o regresión.

- **Aprendizaje no supervisado**. El aprendizaje no supervisado, al contrario que el supervisado, no dispone de ejemplos de salida sino que únicamente dispone de ejemplos de entrada. El aprendizaje no supervisado puede servir para estructurar datos, como por ejemplo en las técnicas de *clustering*. Algunos de los problemas más comunes en ML son la clasificación, donde se trata de conseguir que una máquina consiga clasificar unos datos en concreto, o la regresión, donde se trata de conseguir que la máquina aprenda a establecer una relación continua entre sus datos de entrada y su salida.

- **Aprendizaje por refuerzo**. El aprendizaje por refuerzo es una tipo de ML que busca determinar la mejor acción a realizar en función de un estado. Un **agente** aprende a tomar decisiones según el estado en base a una retroalimentación producida por dichas acciones, traducida como recompensas, tratando de maximizar esta recompensa en el tiempo.

2.2 Redes neuronales artificiales

Las redes neuronales artificiales (*Artificial Neural Networks* o ANN) son un tipo de algoritmos de ML inspirados en el funcionamiento de las neuronas biológicas. Se organizan por capas y están compuestas por una capa de entrada, una de salida y una o más capas ocultas.

Las capas están compuestas por **neuronas**, las cuales son unidades de procesamiento interconectadas entre sí que producen una salida a partir de una entrada en función de sus parámetros (llamados también pesos). Las neuronas a su vez pueden tener una función de activación que se aplica a la salida de estas, y que decide cómo una neurona se activa en función de sus pesos y entradas. Hay muchas funciones de activación distintas. Algunas de las más frecuentes son:

- **Sigmoide** (2.1). Genera una salida en el rango $[0, 1]$. Se suele utilizar en la salida de las redes en problemas de clasificación bianria ya que se puede utilizar como representación de una distribución binomial.

$$\text{Sigmoide}(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

- **Softmax** (*función exponencial normalizada*) (2.1). Genera un vector de K dimensiones normalizado de valores en el rango $[0, 1]$ a partir de un vector de las mismas dimensiones de valores arbitrarios. Se suele utilizar en la salida de resdes en problemas de clasificación multiclase.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{2.2}$$

- **Tanh** (tangente hiperbólica) (2.3). Genera una salida en el rango $[-1, 1]$ calculando la tangente hiperbólica de la entrada.

$$\tanh(x) = \frac{2}{1 + e^{-x}} - 1 \tag{2.3}$$

- **ReLU** (*Rectified Linear Unit*) (2.4) (Hahnloser y cols., 2000). Para los valores negativos resulta en cero mientras que para los positivos en la identidad. Actualmente es la activación más usada en las capas intermedias de una red. Entre sus principales características encontramos que tiene un coste computacional muy bajo, hecho que mejora el rendimiento de la red. Además, ayuda a converger rápidamente a las redes neuronales. Hay distintas variantes de la ReLU como la **LeakyReLU** (2.5) (Maas y cols., 2013) o la **ELU** (2.6) (Clevert y cols., 2015).

$$\text{ReLU}(x) = \max(0, x) \tag{2.4}$$

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\alpha * x), & \text{if } x < 0 \end{cases} \tag{2.5}$$

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (e^x - 1), & \text{if } x \leq 0 \end{cases} \tag{2.6}$$

Las capas de las redes neuronales pueden ser de muchos tipos distintos. Las más comunes son las *capas totalmente conectadas* en las que cada una de las neuronas está conectado con todas las entradas que dispone la capa. Otros tipo de capa muy usado son las *capas*

convolucionales, las cuales se utilizan para la extracción de características de una imagen o las neuronas recurrentes, que se utilizan para el procesamiento de secuencias.

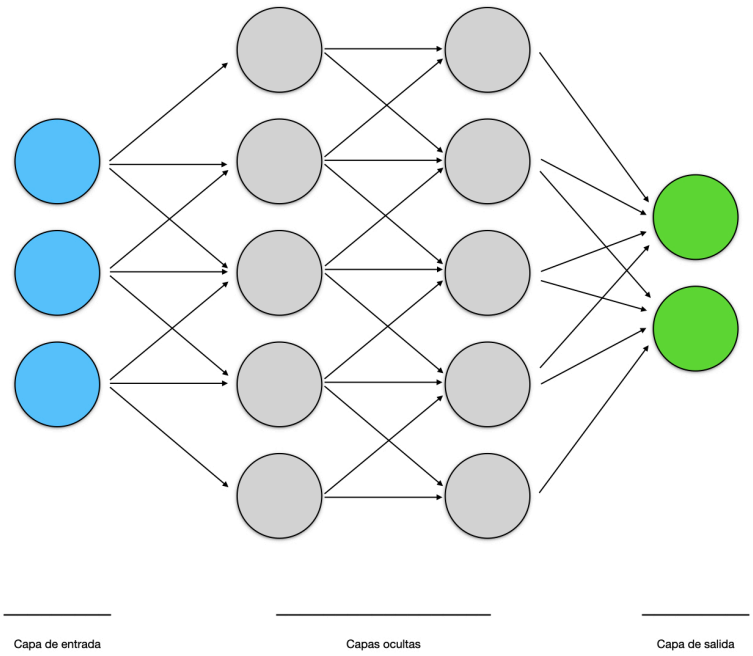

Figura 2.1: Representación gráfica de una red neuronal con 2 capas ocultas.

El entrenamiento de las ANN consiste en modificar los pesos de las neuronas para conseguir que produzca la salida deseada a partir de una entrada. Este entrenamiento se abarca como un problema de optimización. Para ello se minimiza una función de pérdida elegida específicamente para cada problema que determinará el error de la salida de nuestra red. Este error es propagado a través de la red mediante el algoritmo de *back-propagation* (Rumelhart y cols., 1986), que calcula el gradiente de la función de pérdida con respecto a los pesos de cada neurona, y se utiliza una función de optimización para actualizar los pesos a partir de este gradiente. Las funciones de optimización utilizadas son distintas variantes del algoritmo de descenso por gradiente. Algunos ejemplos de estas funciones de optimización son el SGD (Robbins y Monro, 1951), ADAM (Kingma y Ba, 2014) o RMSProp (Tieleman y Hinton, 2012).

2.2.1 Redes neuronales convolucionales

Las redes convolucionales (*Convolutioinal Neural Networks*, CNN) son un tipo de red neuronal especialmente diseñadas para trabajar con imágenes y que han demostrado funcionar mejor que las redes neuronales convencionales para este tipo de tareas. La principal característica de una red convolucional es que disponen de **capas convolucionales** que funcionan a modo de extractor de características y transforman la imagen a una representación más fácil de procesar por las *capas totalmente conectadas*. A diferencia de las anteriores, las CNN

toman como entrada una matriz tridimensional de tamaño $H \times W \times C$[1] en lugar de un vector unidimensional. Esto permite capturar dependencias espaciales en la imagen aplicando los filtros correspondientes.

Las *capas convolucionales* están compuestas por un filtro o *kernel* de tamaño $M \times N$ que es el encargado de llevar a cabo la convolución. Esta se aplica a lo largo de toda la imagen desplazándose mediante un paso o *stride*. Durante la fase de entrenamiento, las variables que se pretenden optimizar en una capa convolucional son los valores de este filtro.

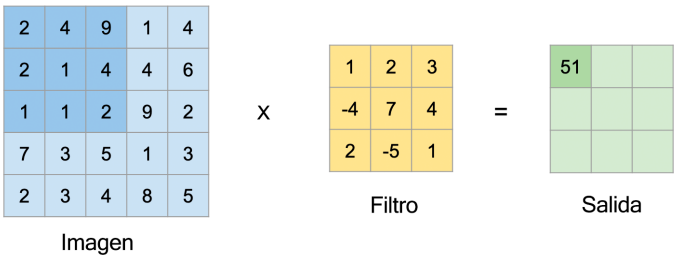

Figura 2.2: Ejemplo de aplicación de un filtro convolucional de tamaño 3×3 a una imagen con *stride* de valor 1.

Además de las capas convolucionales, en las CNN se utilizan capas de ***pooling***. Estas capas se encargan de reducir el tamaño de sus entradas aplicando un operación sobre regiones de la imagen de un tamaño dado $M \times N$, produciendo una salida que dispondrá de un único valor para cada región de tamaño $M \times N$ de su entrada. La capa de *pooling* más común es el *max-pooling* que produce como salida el valor máximo de cada región. El *pooling* tiene 3 funciones principales:

- **Disminuir la potencia computacional necesaria para entrenar la red**. Al reducir el tamaño de las imágenes estamos también reduciendo la cantidad de convoluciones que se tienen que aplicar en cada capa convolucional y también reduciremos el tamaño del vector de características resultante de las convoluciones por lo que las *capas totalmente conectadas* del final de la red tendrán menos parametros.

- **Extraer las características más relevantes de la imagen**. Al simplificar las imágenes forzamos a la red a mantener las características más relevantes de cada imagen, desechando las menos importantes.

- **Extraer características de más alto nivel**. Como estamos reduciendo el tamaño de las imágenes estamos permitiendo que un filtro de una capa convolucional se aplique sobre regiones espacialmente lejanas sin necesidad de aplicar filtros excesivamente grandes. De esta manera favorecemos que la red pueda relacionar distintas regiones de la imagen extrayendo características más generales de la imagen.

[1] Siendo H la altura de la imagen, W el ancho y C el número de canales (por ejemplo, 3 en el caso de las imágenes RGB).

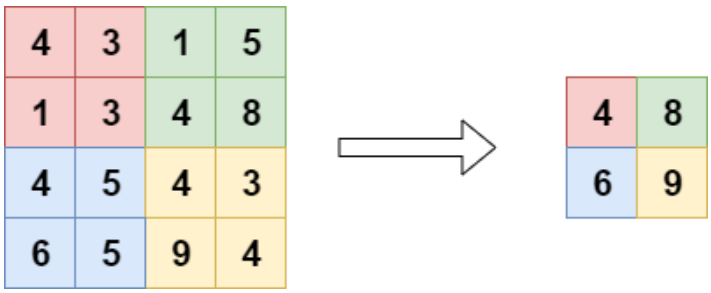

Figura 2.3: Ejemplo de aplicación de una capa *max-pooling* de tamaño 2×2 sobre una matriz de $4 \times 4 \times 1$.

Otro tipo de capas muy comúnmente utilizadas en las CNN son el *dropout* (Srivastava y cols., 2014) que fuerza a la red a anular la activación de una porción de sus neuronas únicamente en la fase de entrenamiento con el objetivo de reducir el sobre-entrenamiento (*overfitting* en inglés) [2] o el *batch normalization* (Ioffe y Szegedy, 2015) que aplica un escalado y ajuste de las activaciones de una red reduciendo el desplazamiento de la covarianza y aportando mayor estabilidad a la red y facilitando el entrenamiento.

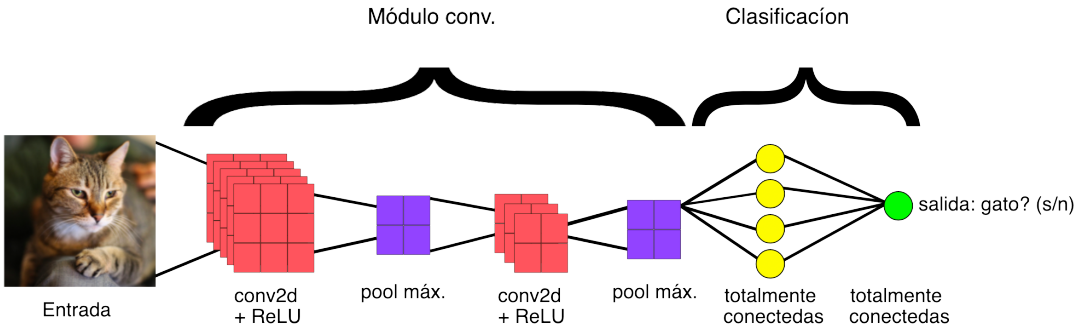

Figura 2.4: Ejemplo de CNN para clasificación.

En la Figura 2.4 podemos ver un ejemplo de CNN completa utilizada como clasificador. En ella podemos ver 2 bloques de capas convolucionales con *pooling* seguidas de 2 capas totalmente conectadas que dan paso a la salida de la red.

2.3 Redes generativas antagónicas (GAN)

Los modelos generativos, como su propio nombre indica, pretenden generar datos aprendiendo la distribución de probabilidad de unos datos. Por ejemplo, un modelo generativo que genere imágenes de gatos aprenderá la distribución de probabilidad de las imágenes de gatos,

[2] Se produce un sobre-entrenamiento o *overfitting* cuando una red genera muy buenos resultados con los datos con los que se ha entrenado pero no con otros. En este caso la red está *memorizando* los datos de entrenamiento en lugar de aprenderlos.

pudiendo generar así imágenes que pudieran ser de gatos.

Las GAN son un tipo de **modelo generativo** formado por redes neuronales, el cual propone la idea de poner a dos redes a competir entre sí (Goodfellow y cols., 2014). Una de las redes, el **generador**, se encarga de producir imágenes a partir de un ruido aleatorio mientras que la segunda, el **discriminador**, trata de diferenciar las imágenes reales de las falsas. A lo largo del entrenamiento se intenta llegar a un equilibrio en el que el discriminador no logre diferenciar entre las imágenes generadas por el generador y las imágenes reales.

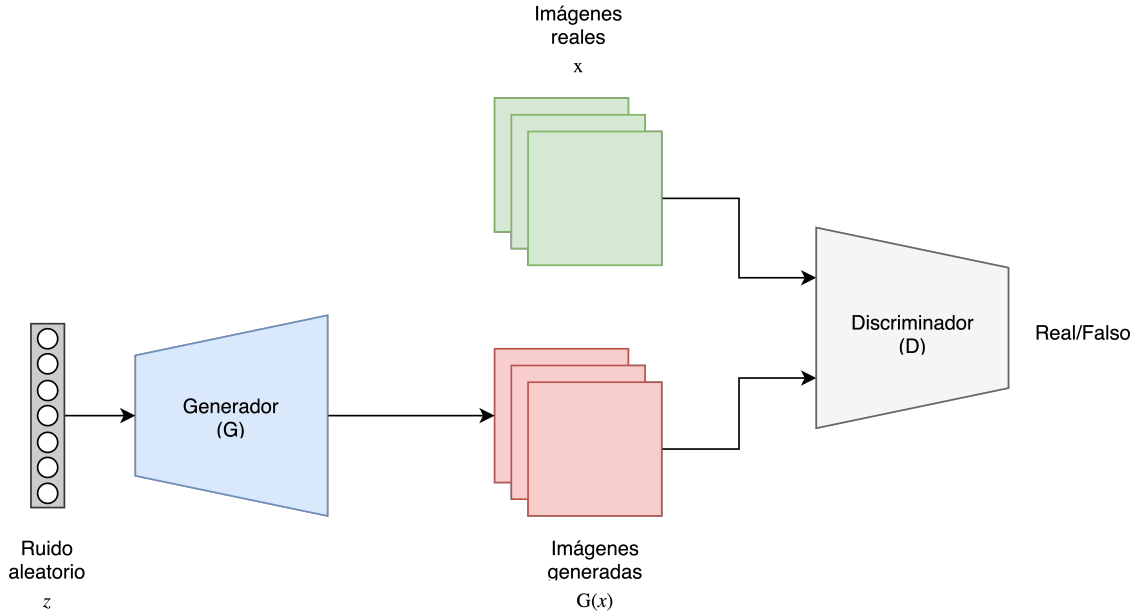

Figura 2.5: Arquitectura básica de una GAN.

Para el entrenamiento de las GAN necesitamos únicamente un conjunto de datos de imágenes de lo que pretendamos generar. En cada iteración, el Generador (G) genera un *lote* de imágenes falsas $G(z)$ a partir de un vector z de ruido aleatorio y entrenaremos el Discriminador (D) con este lote además de con otro de imágenes reales x. El entrenamiento del discriminador se realiza como el de una CNN normal para que distinga imágenes reales y generadas. Para entrenar el generador, propagaremos el error de las imágenes generadas del discriminador al generador y aplicaremos la función de optimización en función de este error. De esta manera, no entrenamos el generador directamente con lo que pretendemos que produzca si no que aprende directamente aquello que necesita para conseguir *engañar* al discriminador.

Es muy común en las GAN utilizar **convoluciones transpuestas** para aumentar el tamaño de la imagen que se va a producir. Las convoluciones transpuestas pueden producir una salida de mayor tamaño que su entrada, funcionando así como función de *upsampling*. Las convoluciones transpuestas se pueden realizar añadiendo un *padding* a cada elemento de la entrada y aplicándole una convolución posteriormente como se puede apreciar en la figura 2.6.

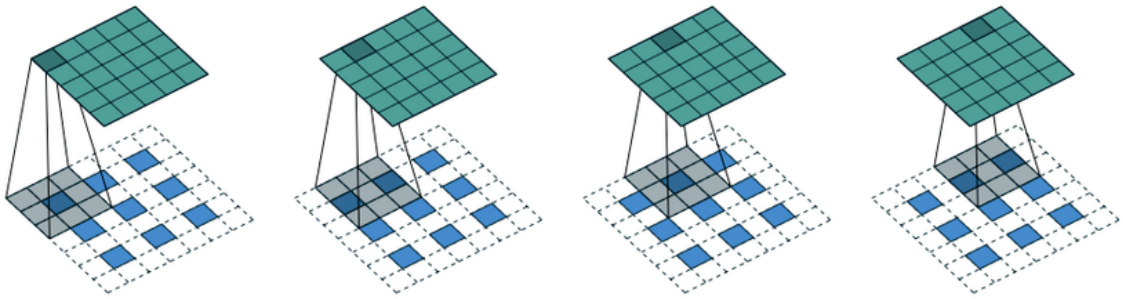

Figura 2.6: Ejemplo de funcionamiento de una convolución transpuesta.

3 Estado del arte de las GAN

Las GAN fueron presentadas en 2014 por un equipo de la Universidad de Montreal dirigido por Ian Goodfellow (Goodfellow y cols., 2014). A partir de entonces han surgido multitud de tipos de GAN distintas capaces de conseguir todo tipo de tareas cada vez más complejas.

Figura 3.1: Ejemplos de imágenes obtenidas de la GAN sacadas de la publicación original (Goodfellow y cols., 2014) con MNIST (`http://yann.lecun.com/exdb/mnist/`) (a) y con TFD (*Toronto Faces Dataset*) (b). En amarillo aparecen las muestras utilizadas para el entrenamiento que más se parecen a las salidas obtenidas para demostrar que la red generadora no memorizan las imágenes que ven durante el entrenamiento.

3.1 CGAN

Mientras que las GAN originales producen una salida a partir de ruido aleatorio, las CGAN (Mirza y Osindero, 2014) (*Conditional Adversarial Network*) pretenden ir un paso más allá y conseguir poder controlar la salida producida por el generador. Lo que se propone es poder condicionar la salida del generador a partir de la introducción de algo de información extra sobre lo que se pretende generar como podría ser una clase u otro tipo de información como pudieran ser características del mismo.

El generador de la CGAN, a parte del ruido aleatorio z se alimenta de una segunda entrada y con la información de lo que se va a generar. Por otra parte, el discriminador también es entrenado con esta etiqueta para no sólo determinar si la entrada que recibe es real o falsa sino que también determina si esta corresponde con la etiqueta y que se le proporciona. De esta manera la el generador aprende a condicionar su salida en función de una segunda entrada y permitiéndonos controlar la salida de este.

Esta arquitectura permite también la generación de contenido más realista en conjuntos de datos multiclase haciendo que la distribución que aprende a generar la red se ajuste a

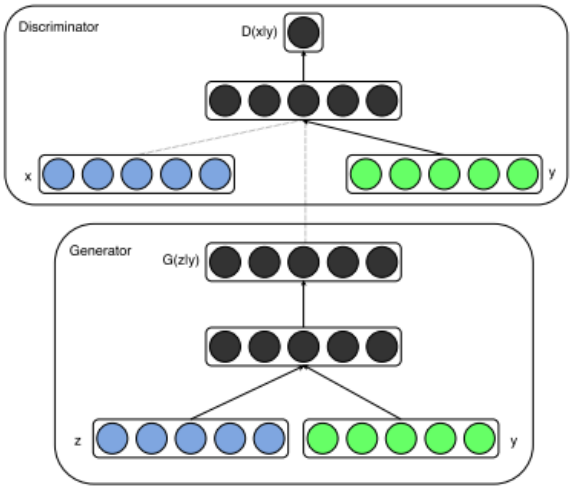

Figura 3.2: Arquitectura de una CGAN. Extraída de la publicación original.

cada una de las clases disponibles en ésta. Por ejemplo, en la figura 3.3 podemos ver un ejemplo de aplicación de esta arquitectura con la base de datos MNIST. Gracias a que la red aprende también la clase de cada uno de los dígitos estamos evitando que la red pueda generar imágenes que parezcan números manuscritos pero que en realidad no correspondan a ningún dígito.

Figura 3.3: Ejemplo de una CGAN entrenada con MNIST condicionando la clase. Cada columna está obtenida condicionando por una clase distinta. Extraída de la publicación original.

3.2 DCGAN

Las DCGAN (*Deep Convolutional Adversarial Networks*) (Radford y cols., 2015) presentan un nuevo diseño en la arquitectura de las GAN. Supusieron un gran avance en la generación de imágenes consiguiendo mejorar el estado del arte. Las DCGAN proponen una simple arquitec-

tura compuesta principalmente por capas convolucionales diseñada con unas características concretas las cuales los autores de la publicación demuestran de manera empírica que ayudan a obtener mejores resultados en las imágenes generadas.

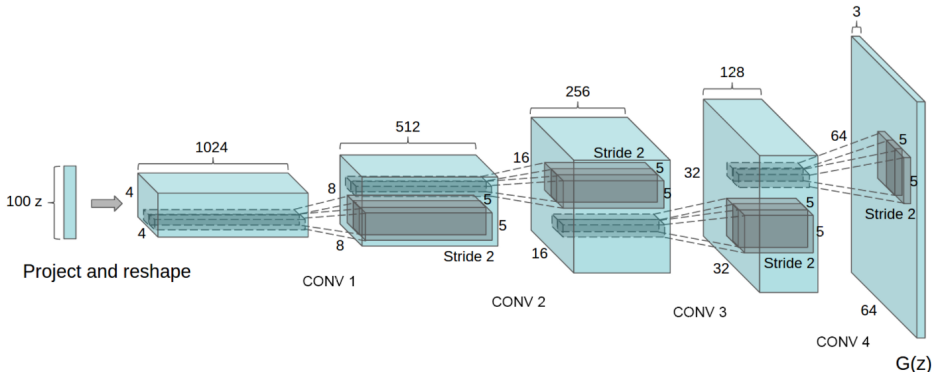

Figura 3.4: Representación gráfica de la arquitectura del generador de una DCGAN. Extraída de la publicación original.

Las características de las DCGAN se resumen en los siguientes puntos:

- Reemplazar todos los *max-pooling* por convoluciones con *stride*.
- Utilizar *batch normalization* en el generador y el discriminador.
- Eliminar las capas totalmente conectadas.
- Utilizar la activación *ReLU* en todas las capas del generador excepto en la salida, que se utiliza *tanh*.
- Utilizar la activación *Leaky ReLU* en todas las capas del discriminador.

En la figura 3.5 vemos un ejemplo de imágenes de habitaciones generadas con una DCGAN. Se puede ver a simple vista que los resultados son más realistas que los vistos anteriormente además de más complejos.

3.3 SketchyGAN

La *SketchyGAN* (Chen y Hays, 2018) se diseñó con el objetivo de sintetizar imágenes arbitrarias a partir de bocetos (*sketch* en inglés). En ella se presentó una arquitectura específicamente diseñada para que la salida esté fuertemente influenciada por la entrada de la red y así conservar el detalle que se esté plasmando en el boceto. Para ello introducen los bloques MRU (*Masked Residual Unit*) diseñados para esta tarea (figura 3.6). Los bloqeus MRU utilizan una máscara interna que aprende durante el entrenamiento para extraer características de manera selectiva de la imagen de entrada que combina con los mapas de características que ha calculado la red neuronal en las capas previas.

El generador recibe las siguientes entradas:

Figura 3.5: Ejemplos de habitaciones generadas con una DCGAN. Extraída de la publicación original.

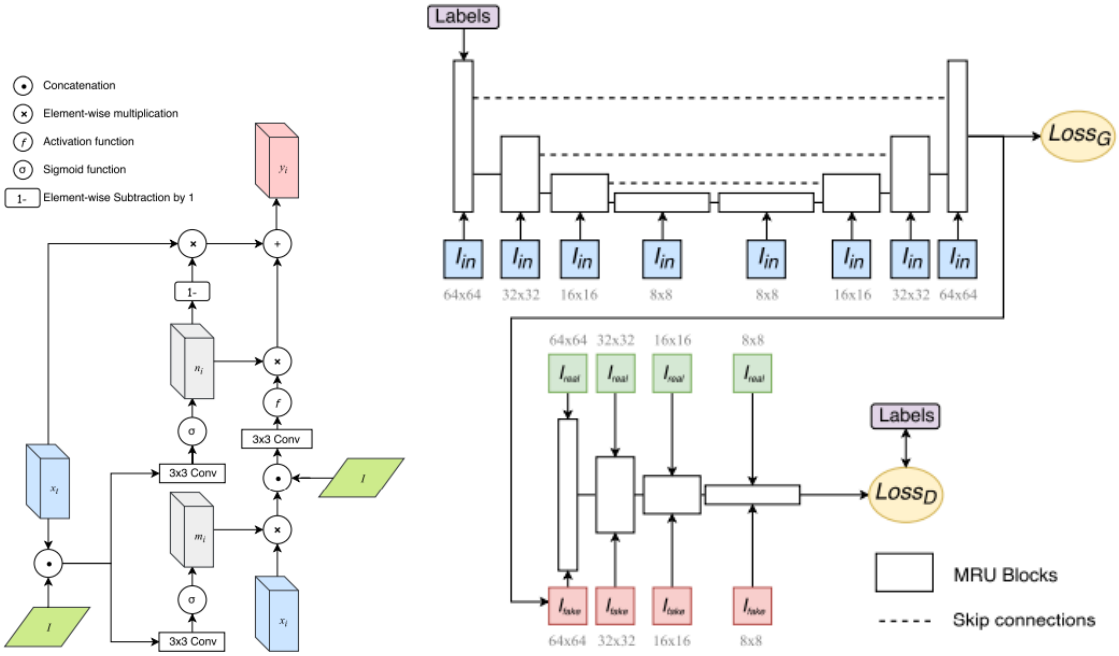

Figura 3.6: A la izquierda, un bloque MRU utilizado por la *SketchyGAN*. A la derecha, arquitectura de la *SketchyGAN*. Imágenes extraídas de la publicación original.

- El **boceto** a partir del cual se generará la imagen.
- Un vector de **ruido aleatorio** que permitirá generar distintas muestras a partir de un mismo boceto.
- Una **etiqueta** que represente la clase del boceto. Esto permitirá a la red saber que es la imagen que está sintetizando y podrá facilitará la generación.

De la misma manera, el discriminador además de determinar si la muestra es real o no, aprenderá a determinar si una muestra que recibe corresponde con la etiqueta asociada, de la misma manera que la CGAN (sección 3.1).

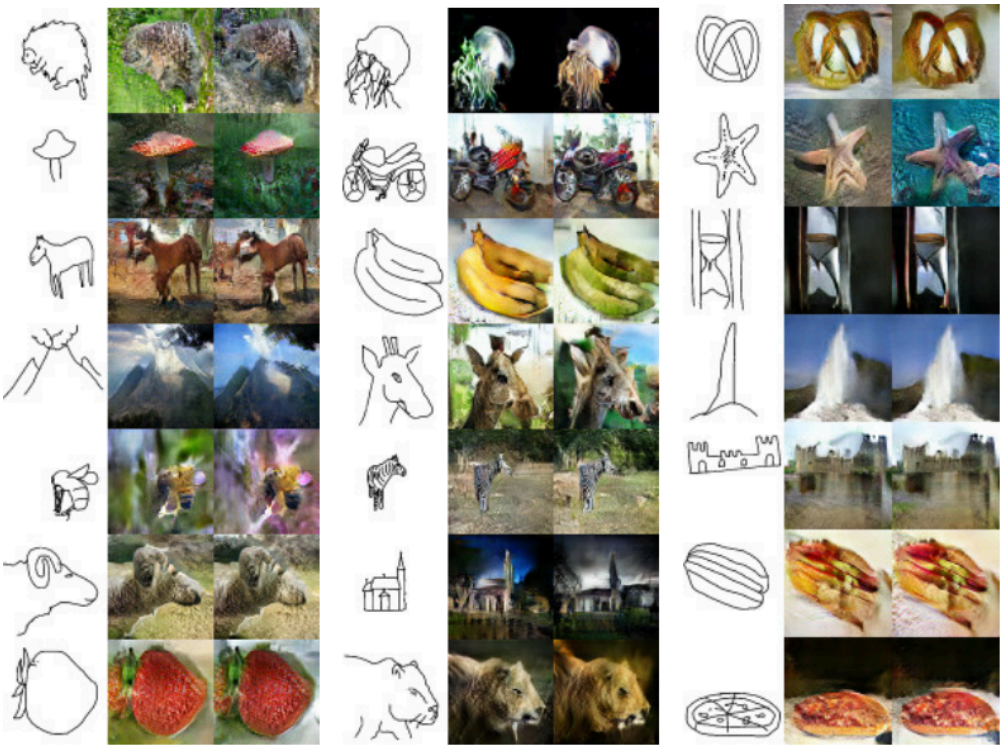

Figura 3.7: Ejemplo de dos salidas distintas a partir de un boceto. Extraída de la publicación original.

Como se puede ver en la figura 3.7, la *SketchyGAN* permite obtener resultados bastante realistas a partir del boceto que recibe como entrada sin perder el detalle que hay en él pudiendo generar además distintas versiones.

3.4 Otros tipos de GAN

A continuación se muestran otros ejemplos de lo que pueden conseguir las GAN para completar el análisis de su estado de la cuestión.

La **SRGAN** (*Super Resolution Generative Adversarial Network*) (Ledig y cols., 2016) permite aumentar la resolución de imágenes. Mientras el discriminador intenta determinar si una imagen está en su resolución original o ha sido redimensionada por el generador, el generador aprende a aumentar el tamaño de imágenes de baja resolución de forma que las imágenes resultantes sean más nítidas que si utilizáramos los métodos de interpolación tradicionales. Cabe destacar que los píxeles que el generador añade a la imagen resultante no corresponden necesariamente con el contenido original de la imagen, ya que es un dato que no está disponible en la imagen y la red no puede conocer pero sí son coherentes con el resto de la imagen.

La ***CycleGAN*** (Zhu y cols., 2017) se presenta como un *traductor imagen-imagen* que

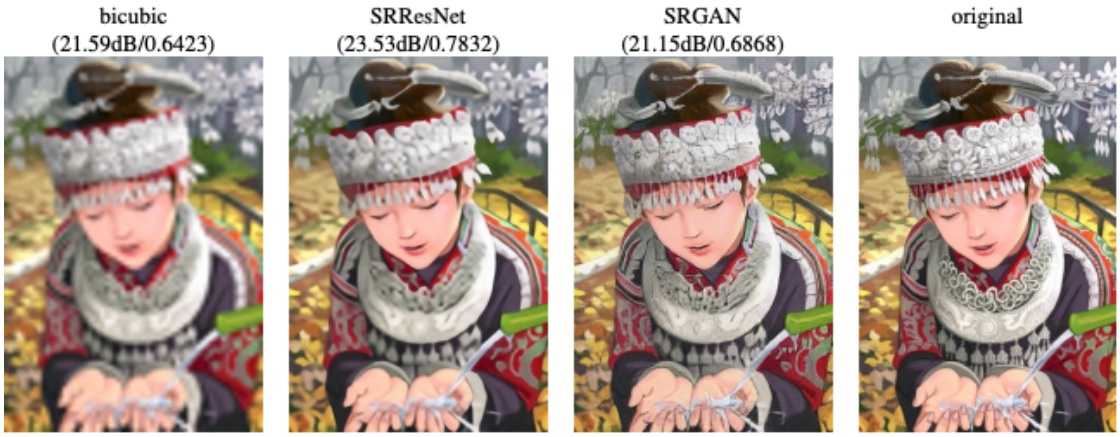

Figura 3.8: A la izquierda, una imagen redimensionada con interpolación bicúbica. En el centro, la misma imágen redimensionada con dos tipos distintos de SRGAN. A la derecha, la imagen original. Extraída de la publicación original.

permiten establecer una relación entre dos tipos de imágenes distintas, A y B, y realizar una *traducción* entre ellas. Por ejemplo, son capaces de cambiar el color del pelo de una persona en una imagen o estilizar una imagen para hacer parecer que es una pintura. Las CycleGAN se componen por dos GAN independientes, cada una de las cuales se encarga de realizar una traducción: una de ellas convierte imágenes de tipo A al tipo B y la otra del tipo B al tipo A. En el entrenamiento de la CycleGAN se utilizan imágenes de las dos clases distintas, A y B, las cuales se pretenden convertir a la clase contraria y además poder volver a convertirse posteriormente a su clase original. En la figura 3.9 podemos ver algunos ejemplos de aplicación de la CycleGAN con distintos tipos de imágenes.

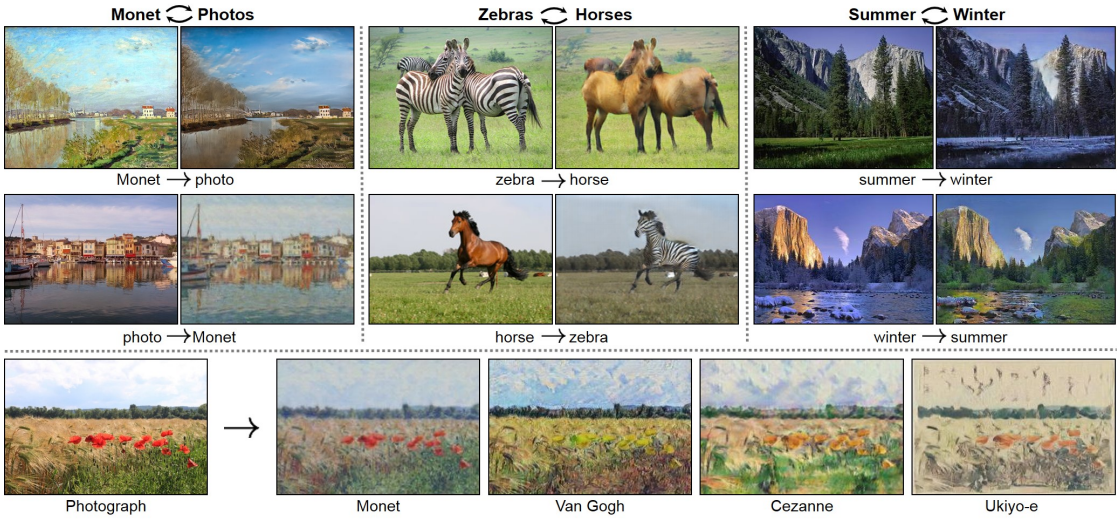

Figura 3.9: Distintos ejemplos de aplicación de la *CycleGAN*. Extraída de la publicación original.

Otro tipo de GAN es la ***StyleGAN*** (Karras y cols., 2018). Con la StyleGAN se consiguieron

generar imágenes de caras ficticias de personas tan realistas que son imperceptibles por el ojo humano, permitiendo incluso especificar características de la cara que queremos generar. La red generadora recibe como entrada una codificación de 40 características (como género, color o tipo de pelo, etnia...) en un vector z, en un espacio latente que permite a la red interpolar las características que recibe como entrada. En la figura 3.10 podemos ver algunos ejemplos arbitrarios de imágenes generadas mediante la StyleGAN.

Figura 3.10: Ejemplos de imágenes generadas con la StyleGAN. Extraída de la publicación original.

4 Metodología

En este capítulo se define la metodología utilizada para el desarrollo del proyecto. Primero se va a proceder a presentar el enfoque, donde se expondrá las arquitectura diseñada para la implementación de las GAN y, además, se detallará el algoritmo de entrenamiento. Posteriormente se hablará de las herramientas utilizadas para el desarrollo del proyecto, así como el conjunto de datos de logotipos reales utilizado para el ajuste de parámetros (entrenamiento) de las GAN.

4.1 Enfoque

En esta sección se detalla el enfoque de la solución que se presenta para la realización del sistema de generación de imágenes a partir de bocetos. Este sistema, formado por unas GAN, a su vez formada por dos redes neuronales —el generador y el discriminador— para las cuales se detalla su arquitectura a continuación. Además, se necesita implementar un algoritmo de entrenamiento para estas redes que también se expone posteriormente.

4.1.1 Arquitectura

Para la implementación del proyecto necesitamos diseñar las arquitecturas que conformarán las redes neuronales de las que se componen las GAN. Para ello necesitamos definir primero qué arquitectura tendrán estas redes: el generador y el discriminador.

En base al estado de la cuestión de las GAN visto en el capítulo 3, se han diseñado dos arquitecturas distintas para el generador y una para el discriminador. Estas arquitecturas se han diseñado de manera ajustable, para así poder experimentar con distintos hiperparámetros.

Las imágenes de logos se han tratado con un tamaño de 64×64 en formato RGB, mientras que los bocetos se han representado mediante una máscara del mismo tamaño.

4.1.1.1 Generador

Para el generador se han diseñado dos arquitecturas diferentes. Ambas reciben como entrada una máscara del boceto con los valores 0 y 1 con dimensiones de $64 \times 64 \times 1$ y devuelven como salida una imagen RGB de dimensiones $64 \times 64 \times 3$ con valores en el rango $[-1, 1]$. Ambas arquitecturas se han diseñado con el objetivo de maximizar la influencia del boceto en la salida y que esta se parezca lo máximo posible a su entrada.

En la **primera arquitectura** se ha tratado de que la salida de modifique lo mínimo posible su entrada. Por ello, se ha evitado que se realicen cambios de alto nivel en la imagen para lo que no se han utilizado *poolings* ni se han aplicado *strides* mayores a 1. De esta manera, al aplicarse los filtros convolucionales, siempre actúan de manera local. Como todas las transformaciones se aplican localmente, los cambios que se producen **no son excesivamente**

agresivos por lo que no se modifica en gran mediada la entrada, aplicándose únicamente pequeñas transformaciones como coloraciones o contorneados,

La arquitectura se basa en bloques convolucionales aplicados de manera secuencial y un bloque de salida. Los bloques convolucionales están formados por una capa convolucional con *stride* = 1, una capa de *batch normalization* y una activación *Leaky ReLU* en la salida del bloque. El tamaño del kernel de las capas convolucionales está parametrizado así como el número de filtros de la primera capa. Para cada bloque i, el número de filtros que tiene su capa convolucional es de 2^i. El bloque de salida se compone de una capa convolucional con 3 filtros para generar la salida de 3 canales y una activación *tanh* (tangente hiperbólica) para normalizar la salida.

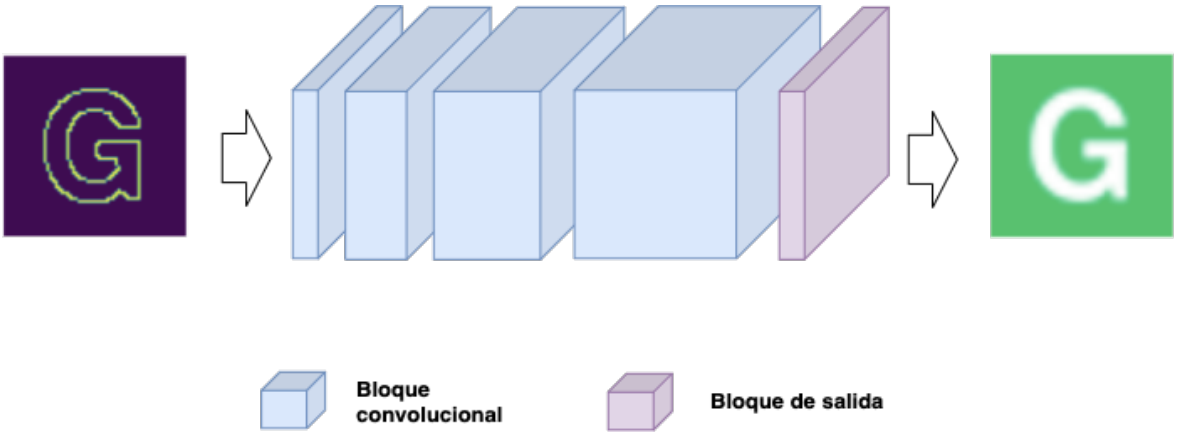

Figura 4.1: Representación gráfica de la arquitectura del *generador 1*.

La **segunda arquitetura** se basa en las arquitecturas de los *autoencoders*, de manera similar a la Sketchy-GAN vista anteriormente (capítulo 3). Los *autoencoders* se componen de dos partes, el *enconder* el cual se basa en extraer y codificar las características más relevantes de una imagen y el *decoder*, que pretende regenerar la entrada a partir de estas características codificadas. El encoder y el decoder suelen poseer arquitecturas simétricas, donde el encoder primero reduce las dimensiones de la entrada y el decoder las aumenta. En este caso, la arquitectura se compone de estas mismas dos partes, la primera se encargará de de extraer las características a alto nivel del boceto y la segunda, generará la imagen deseada a partir de este boceto. Para aumentar la influencia de la imagen de entrada y que la salida la tenga más en cuenta así como de las características de más bajo nivel, a la red se le añaden unas conexiones residuales de cada capa del encoder con su homóloga del decoder. De esta manera, por una parte conseguimos que la red genere la salida en base a las características más básicas y primitivas de la imagen conservando líneas, puntos y formas por una parte, y por otra a características de más alto nivel pudiendo así contextualizar el boceto y sintetizarlo acorde a sus características.

La arquitectura está compuesta por tres bloques distintos: el bloque encoder, el bloque decoder y el bloque de salida. Los bloques del encoder se componen de una capa convolucional, una capa de *batch normalization* y una activación *LeakyReLU*. Los bloques se agrupan en grupos de n bloques, al final de los cuales se realiza un *downsampling* asignándole a la capa convolucional un stride de 2. Este parámetro n es un hiperparámetro del generador. Los blo-

ques del decoder son iguales que los bloques del encoder a excepción de la capa convolucional, que en este caso se utiliza una convolución transpuesta. Estos bloques se agrupan a su vez en grupos de n. Por otra parte, a la entrada de cada grupo de bloques se le concatena la salida de su grupo homólogo del encoder como conexión residual. Finalmente, el bloque de salida se compone de una convolución transpuesta con 3 filtros y una activación *tanh* para generar la imagen de salida.

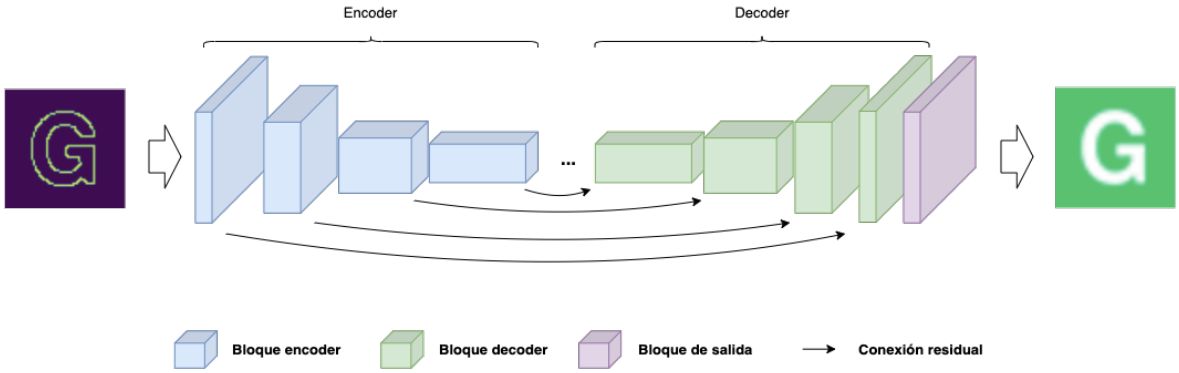

Figura 4.2: Representación gráfica de la arquitectura del *generador 2*.

4.1.1.2 Discriminador

Para el discriminador se ha diseñado una arquitectura basada en DCGAN, siguiendo algunas de las recomendaciones que se indican en el artículo:

- No se han utilizado capas totalmente conectadas.
- Se han utilizado convoluciones con un *stride* de 2 en lugar de *max-poolings*.
- *Batch normalization* en todas las capas.
- Se ha utilizado *Leaky ReLU* en todas las capas intermedias de la red.

La red recibe como entrada un tensor de dimensiones $(64, 64, 3)$ escalado al rango $[-1, 1]$ y como salida devuelve un valor en el rango $[0, 1]$ que representa la probabilidad de que la entrada sea real, en el caso de que esté cercano a 0, o falso en el caso contrario.

Esta arquitectura se compone de bloques convolucionales y de un bloque de salida. Los **bloques convolucionales** están compuestos por una capa convolucional con $stride = 2$, una capa de *batch normalization* y una activación *Leaky ReLU* en la salida del bloque. Se parametriza el tamaño del kernel de las capas convolucionales y el número de filtros de la primera capa. Para cada bloque i, el número de filtros que tiene su capa convolucional es de 2^i. El **bloque de salida** está compuesto por una capa convolucional de $kernel\_size = 1$ y una función sigmoide como activación. Para evitar las capas totalmente conectadas, se debe conseguir que la salida del último bloque tenga unas dimensiones de $1 \times 1 \times 1$. Para ello, se aplican tantos bloques convolucionales como sea posible para reducir el tamaño de la imagen a estas dimensiones. Al tener un *stride* de 2, la salida de cada bloque tienen un ancho y alto

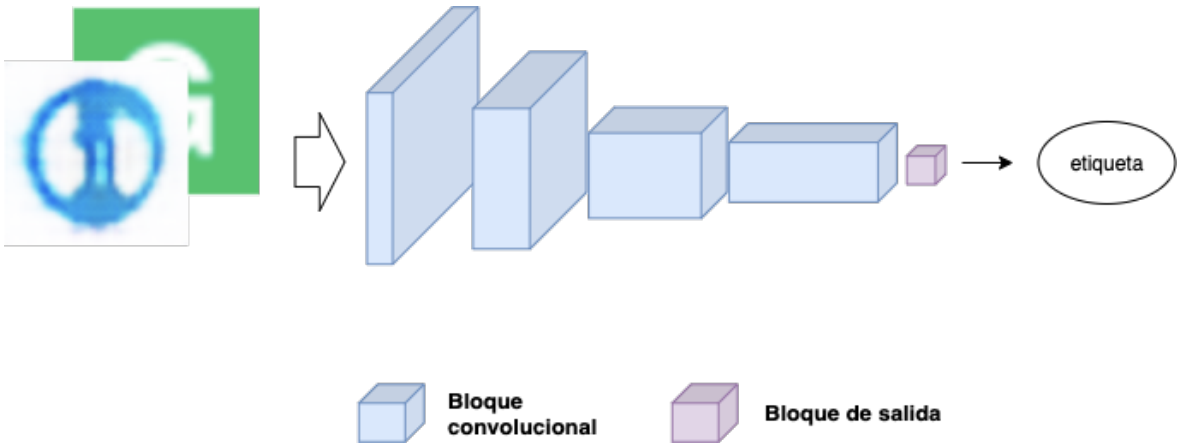

Figura 4.3: Representación gráfica de la arquitectura del discriminador.

la mitad del de su entrada. Para obtener la salida esperada, la red tendrá $log_2(S)$ bloques convolucionales, siendo S el tamaño de la imagen. En nuestro caso, al tener como entrada una imagen de 64×64, tendremos 6 bloques convolucionales.

4.1.2 Entrenamiento

El punto más importante en el entrenamiento, y la base del funcionamiento de las GAN, es la función de pérdida. Ésta se encarga de propagar el error del discriminador al generador y conseguir que éste aprenda a generar imágenes que el discriminador no pueda clasificar correctamente. Las funciones de pérdida del discriminador son las siguientes:

$$Loss(D(x)) = BCE_c(D(x)) \tag{4.1}$$

$$Loss(G(x)) = BCE_{\overline{c}}(D(G(x))) \tag{4.2}$$

En ambos casos la pérdida utilizada es una función de entropía cruzada binaria (BCE). En el discriminador su función de pérdida $Loss(D(x))$ (4.1) calcula el BCE a partir de sus salida como para cualquier otro caso de clasificación. Para el generador, la función de pérdida $Loss(G(x))$ (4.2) se calcula a partir de la salida del discriminador recibiendo como entrada imágenes generadas por el generador. En este caso, la función BCE que se aplica se hace con las clases invertidas. Así, se calculará el error del discriminador al determinar si las imágenes generadas son reales, este error se propagará a través del generador y por lo tanto, se podrá optimizar de manera que las imágenes que genere sean más reales para el discriminador. De esta manera, mientras el discriminador estará aprendiendo a discriminar las imágenes reales de las generadas, el generador aprenderá a ponérselo difícil al discriminador.

El algoritmo de entrenamiento se basa en el entrenamiento típico de las *gan*. En cada iteración:

- Se obtiene el *batch* de imágenes
- Se generan los bocetos del *batch* con el algoritmo de *canny*

- Se realiza la inferencia $G(x)$ con los bocetos
- Se realiza la inferencia $D(x)$ con las imágenes reales
- Se realiza la inferencia $D(G(x))$ con las imágenes generadas
- Se calcula la pérdida $Loss(D(x))$ con la salida de $D(x)$ y $D(G(x))$
- Se calcula la pérdida $Loss(G(x))$
- Se calcula los gradientes del error propagado de $Loss(D(x))$ y a través de D
- Se calcula los gradientes del error propagado de $Loss(G(x))$ a través de D y G
- Se actualizan los pesos de D a partir del gradiente de $Loss(D(x))$
- Se congelan los pesos de D y actualizan los pesos de G a partir del gradiente de $Loss(G(x))$

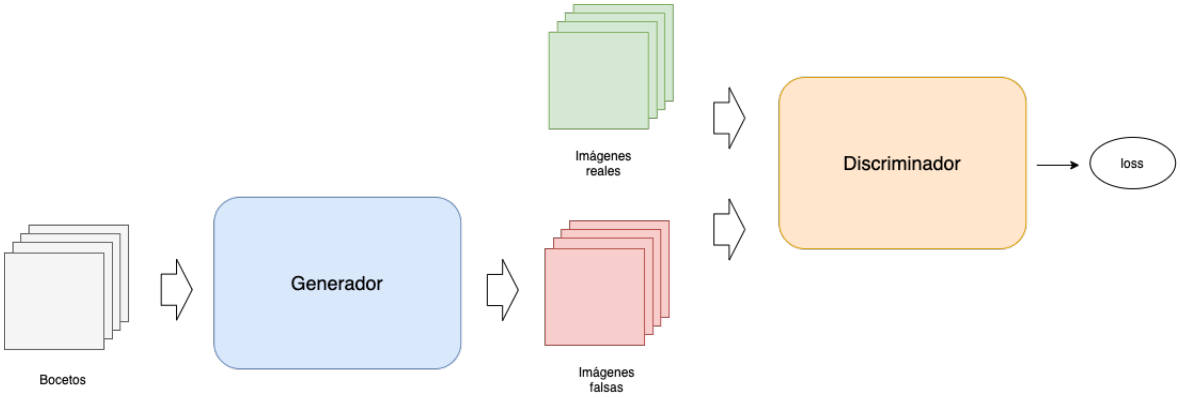

Figura 4.4: Diagrama del entrenamiento.

4.1.2.1 Reconstrucción de bocetos

Uno de los objetivos más importantes del sistema es que la imagen que se genere guarde relación con la entrada del generador, el boceto. Pese que a las arquitecturas seleccionadas pretenden que el generador base la generación de la imagen en el boceto de entrada en la mayor medida posible, nada garantiza que esta salida tenga algo que ver con la entrada. Ya que la pérdida se calcula únicamente en función de la salida del discriminador, el generador solo puede aprender a generar imágenes que el discriminador determine como reales y no que estas se basen en la entrada del generador.

Para solucionar este problema, se ha diseñado una variante del entrenamiento descrito anteriormente que incorpora un módulo de regeneración del boceto a partir de la salida del generador. Este módulo se presenta como una novedad metodológica que no ha sido utilizada previamente en ningún trabajo conocido, y por lo tanto diseñado de manera específica para este proyecto.

En la fase del entrenamiento, además de hacer que el generador aprenda a generar imágenes que el discriminador determine como reales, también deberá hacer que a partir de estas imágenes, el boceto que se genere se parezca lo máximo posible al boceto origina. Para ello, a la función de pérdida descrita previamente se le incluye una pérdida $Loss(R(x))$ calculada a partir de los bocetos originales y la regeneración de los bocetos a partir de la salida del generador. La función de pérdida resultante, la nueva función de pérdida del generador $Loss(G(x))$ se calcularía mediante una suma ponderada de la pérdida del discriminador y la pérdida del generador de bocetos. La función de pérdida utilizada para el módulo de reconstrucción es la función de entropía binaria cruzada (BCE) aplicada a cada píxel de la reconstrucción.

$$Loss(R(x)) = \sum_{i=1}^{n} \frac{BCE_C(R(x)_i)}{n} \tag{4.3}$$

$$Loss(G(x)) = BCE_{\overline{c}}(D(G(x))) + \lambda Loss(R(G(x))) \tag{4.4}$$

Para la regeneración de bocetos, el caso ideal sería utilizar un filtro *canny*, ya que obtendríamos bocetos muy similares a los que se utilizan como entrada del generador. Por otra parte, este algoritmo **no es derivable** y por lo tanto, el error no se puede propagar mediante *back-propagation* por la red. Como alternativa, se va a utilizar una aproximación basada en una red de una única capa convolucional que tendrá como salida una imagen de las mismas dimensiones que su entrada pero con un único canal. Como esta red no dispondrá de *poolings*, únicamente producirá cambios locales en la entrada y al no ser profunda, no tendrá capacidad suficiente como para alterar la entrada en mayor medida. En cambio, sí será suficiente para obtener los bordes de la imagen, pudiendo obtener resultados similares que con *canny*.

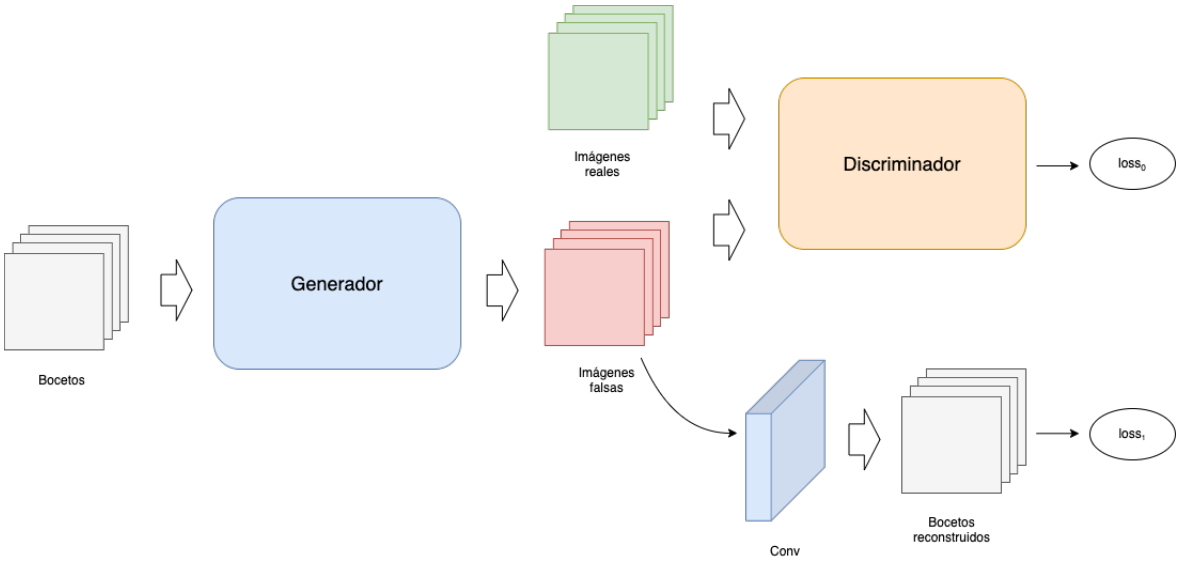

Figura 4.5: Diagrama del entrenamiento con el módulo de reconstrucción de bocetos.

Esta red se entrenará durante el proceso de entrenamiento de la GAN, recibiendo como entrada la salida del generador con el objetivo de que genere la misma imagen que ha recibido

el generador como entrada (el boceto original).

4.2 Herramientas

Para la realización del proyecto se han utilizado las siguientes herramientas:

- **Python**[1] es un lenguaje de programación interpretado de alto nivel. Es un lenguaje muy utilizado para experimentación, en general, y también para *deep learning* y *computer vision*, en particular, dadas sus caractericticas. Por ello, dispone de gran cantidad de herramientas libres para llevar a cabo tareas de este tipo. Se ha utilizado Python para implementar toda la experimentación realizada en el proyecto.

- **Tensorflow**[2] es una librería *open source* desarrollada principalmente por Google en c++ y con un wrapper para Python, orientada al desarrollo de grafos computacionales muy utilizada en *machine learning* y *deep learning*. Tensorflow aporta todas las operaciones a bajo nivel necesarias para la implementación de redes neuronales, además de distintas APIs de más alto nivel que ayudan al desarrollo de estas. En concreto, en este proyecto se ha utilizado la API *tf.data*, que permite una creación sencilla y eficiente de flujos de carga de datos y preprocesamiento, y *keras*, una API que permite la creación y el entrenamiento de redes neuronales de manera muy sencilla y con gran flexibilidad.

- **OpenCV**[3] es una librería *open source* de visión por computador implementada en C++ con *wrappers* para multitud de lenguajes de programación (entre ellos Python). Se ha utilizado esta librería para realizar tareas relacionadas con el procesamiento de imágenes.

- **NumPy**[4] es una librería *open source* para Python de procesamiento científico que permite realizar operaciones con vectores multidimensionales. Provee una interfaz sencilla para hacer todo tipo de operaciones entre estos vectores de manera eficiente y rápida. En este proyecto se ha utilizado NumPy para el tratamiento de datos numéricos y/o imágenes.

- **Matplotlib** [5] es una librería *open source* para Python para visualización de datos. Permite realizar representaciones de todo tipo de datos en gráficas, imágenes, mapas de color, histogramas... Se ha utilizado esta librería principalmente para realizar un análisis visual de los datos que se han utilizado como imágenes utilizadas para el entrenamiento o obtenidas de la red, métricas obtenidas durante el entrenamiento...

- **Google Colab**[6] es una plataforma de Google que ofrece recursos limitados gratuitos para el procesamiento de datos con Python. Se ha utilizado Google Colab para la realización de los experimentos ya que permite el uso de GPUs que reducen significativamente el tiempo de entrenamiento de las redes neuronales.

[1] https://www.python.org/
[2] https://www.tensorflow.org/
[3] https://opencv.org/
[4] https://numpy.org/
[5] https://matplotlib.org/
[6] https://colab.research.google.com/

4.3 Dataset

Uno de los aspectos fundamentales del proyecto es la selección de un **conjunto de datos** o *dataset*, necesario para el entrenamiento del modelo. Se necesita una cantidad considerable de imágenes de aquello que se quiera generar para conseguir que la GAN pueda aprender a generar este tipo de contenido. Para ello, se ha buscado un conjunto de datos de logotipos de empresas.

Large Logo Dataset (LLD)[7](Sage y cols., 2017) es un conjunto de datos de logotipos reales que contiene en total más de 600.000 muestras distintas. Los logos fueron obtenidos por los creadores mediante técnicas de *scrapping* de la lista *Top 1 million sites* de Alexa[8] que contiene las 1.000.000 webs más visitadas. El conjunto de datos está separado en dos:

Figura 4.6: Ejemplos de imágenes de LLD-icon de 32×32 píxeles.

- **LLD-icon** (figura 4.6). Contiene 548.210 imágenes de 32×32 píxeles obtenidos de los *favicons* de las páginas web. Existe además un subconjunto de estas imágenes llamado **LLD-icon-sharp** con 221.369 imágenes que recoge aquellos logos que están representados de manera más nítida, siendo imágenes de mayor calidad que las que no están en este subconjunto.

- **LLD-logo** (figura 4.7. Contiene 122.920 imágenes de alta resolución, de entre 64×64 y 400×400 píxeles, siendo la mayoría de ellas del tamaño máximo. Estas imágenes fueron obtenidas de los perfiles de Twitter[9] de las empresas de estas webs.

Para la realización de la mayoría de experimentos se ha utilizado el subconjunto **LLD-icon-sharp** ya que el tamaño reducido de las imágenes permite realizar los experimentos y aun así contienen el suficiente nivel de detalle para representar toda la información necesaria del logotipo. Por otra parte.

[7]https://data.vision.ee.ethz.ch/sagea/lld/
[8]https://www.alexa.com/topsites
[9]https://twitter.com/

Figura 4.7: Ejemplos de imágenes de LLD-logo de 400 × 400 píxeles.

4.3.1 Generación de bocetos

Además de las imágenes de logotipos se necesitan muestras de bocetos a partir de los cuales se van a sintetizar estos logotipos. No se han encontrado ningún conjunto de datos de bocetos de este tipo así que se ha procedido a generar un conjunto de datos sintéticos de bocetos a partir de las imágenes del conjunto de datos de logotipos. Para esta tarea se ha utilizado el algoritmo *canny* (Canny, 1986) para la detección de bordes, en concreto la implementación de *OpenCV* que permitía generar imágenes bastante similares a las que podría generar una persona manualmente.

Figura 4.8: Ejemplos de pares imagen-boceto de los bocetos generados.

En la figura 4.8 podemos ver unos ejemplos de generaciones de bocetos a partir de unas imágenes del dataset mediante el algoritmo de *canny*. El algoritmo se ha aplicado con los

parámetros por defecto de OpenCV y configurando los umbrales de histeresis de manera dinámica para cada imágen para que se ajuste correctamente a cada una de ellas:

$$umbral_1 = max(mediana(v) \times (1 - \sigma), 0) \tag{4.5}$$

$$umbral_2 = min(mediana(v) \times (1 + \sigma), 255) \tag{4.6}$$

5 Experimentación

En el siguiente capítulo se detalla la experimentación realizada en el proyecto. Primero se expondrán algunos detalles de la implementación de las arquitecturas, luego se comentarán las métricas y métodos de evaluación que se utilizarán en los experimentos, posteriormente se expondrán los experimentos a realizar y luego se mostrarán los resultados.

5.1 Detalles de la implementación

A continuación se realizará una recopilación de los hiperparámetros de las arquitecturas expuestas en la sección 4.1.1.

5.1.1 Discriminador

En la arquitectura del discriminador disponemos de bloques convolucionales y de un bloque de salida.

Bloque Convolucional	
Capas	Parámetros
Conv2D	$kernel$ $filtros$ $stride = 2$ $padding = igual$
BatchNorm	
Dropout	$razón = 30\%$
LeakyReLU	$\alpha = 0.1$

Bloque de Salida	
Capas	Parámetros
Conv2D	$kernel = 1 \times 1$ $filtros = 1$ $stride = 1$ $padding = igual$
sigmoid	

Tabla 5.1: Definición de bloques del discriminador.

En la tabla 5.1 se detallan los bloques del discriminador especificando los parámetros de cada uno de ellos. El **bloque convolucional** dispone de una capa convolucional (*Conv2D*), una capa de *batch normalization* (*BatchNorm*), una capa de *dropout* y una activación *LeakyReLU*. La capa convolucional, tendrá un *stride* con valor 2 en todos los casos y se le aplicará un *padding* de manera que la salida sea igual que su entrada. El valor del tamaño del filtro (*kernel*) y la cantidad de filtros es un hiperparámetro de la arquitectura a configurar en cada experimento. La capa de *dropout* tendrá una razón del 30% (se cancelarán el 30% de las neuronas) y la activación *LeakyReLU* tendrá un valor de α de 0.1. El bloque de salida tiene todos los parámetros fijos. Dispone de una capa convolucional (*Conv2*) con $kernel = 1 \times 1$,

$filtros = 1$, $stride = 1$ y $padding = igual$ y una capa de activación con una función sigmoide (*sigmoid*) para generar una salida en el rango $[0, 1]$.

Discrimiador		
Bloque	Parámetros	Tamaño salida
convolucional_1	$kernel$ $filtros = f_{iniciales} * 2^0$	32×32
convolucional_2	$kernel$ $filtros = f_{iniciales} * 2^1$	16×16
convolucional_3	$kernel$ $filtros = f_{iniciales} * 2^2$	8×8
convolucional_4	$kernel$ $filtros = f_{iniciales} * 2^3$	4×4
convolucional_5	$kernel$ $filtros = f_{iniciales} * 2^4$	2×2
convolucional_6	$kernel$ $filtros = f_{iniciales} * 2^5$	1×1
salida		1×1

Tabla 5.2: Definición de la arquitectura del discriminador.

La arquitectura del discriminador se define en la tabla 5.2. Dispone de $\log_2(64) = 6$ bloques convolucionales con el objetivo de reducir el tamaño de la imagen de entrada a un tensor de 1×1 y ponder generar la salida mediante el bloque de salida. La arquitectura dispone de 2 hiperparámetros: el valor de *kernel*, que define el tamaño del *kernel* de las capas convolucionales cada uno de los bloques convolucionales, y el valor de **filtros iniciales** ($f_{iniciales}$) que define la cantidad de filtros de la primera capa. En las capas sucesivas los filtros son el doble de la capa anterior (en cada capa número n habrán $f_{iniciales} * 2^n$).

5.1.2 Generador 1

El generador 1 está formado por bloques convolucionales y un bloque de salida. En la tabla 5.3 podemos ver una definición de los 2 tipos de bloques distintos del generador 1. El **bloque convolucional** se forma de la misma manera que el bloque convolucional del discriminador descrito en 5.1.1, a diferencia de que no se ha utilizado *dropout*. El **bloque de salida** se forma por una capa convolucional (*Conv2D*) con todos los parámetros fijos: un *kernek* de 4×4, 3 filtros para generar la imagen RGB, un *stride* de 1 y un *paddingigual*. La función de activación del bloque de salida es la función de la tangente hiperbólica (*tanh*).

Como podemos ver en la tabla 5.6, el generador 1 se compone de una cantidad variable de bloques convolucionales definida por el hiperparámetro *depth* y un bloque de salida al final. Al tener todos los bloques un *stride* de valor 1, la salida de todos los bloques es igual a la de la entrada de la red (64×64). De igual manera que en el discriminador, la cantidad de filtros

Bloque Convolucional	
Capas	Parámetros
Conv2D	$kernel$ $filtros$ $stride = 1$ $padding = igual$
BatchNorm	
LeakyReLU	$\alpha = 0.1$

Bloque de Salida	
Capas	Parámetros
Conv2D	$kernel = 4 \times 4$ $filtros = 3$ $stride = 1$ $padding = igual$
tanh	

Tabla 5.3: Definición de bloques del generador 1.

Generador 1		
Bloque	Parámetros	Tamaño salida
convolucional_1	$kernel$ $filtros = filtros_{iniciales} * 2^0$	64×64
convolucional_2	$kernel$ $filtros = filtros_{iniciales} * 2^1$	64×64
convolucional_3	$kernel$ $filtros = filtros_{iniciales} * 2^2$	64×64
	...	
convolucional_n	$kernel$ $filtros = filtros_{iniciales} * 2^n$	64×64
salida		64×64

Tabla 5.4: Definición de la arquitectura del generador 2.

se duplica en cada capa, quedando los filtros de la primera capa configurables mediante el hiperparámetro $f_{iniciales}$. Finalmente, el bloque de salida genera la imagen resultante.

5.1.3 Generador 2

En el generador 2 encontramos 3 tipos de bloques: el bloque del encoder, el bloque del decoder y el bloque de salida. En la tabla 5.5 podemos ver la definición de estos 3 bloques. El bloque del encoder es bastante similar al bloque convolucional del generador 1 definido en 5.1.2, a diferencia de que el **bloque del encoder** dispone del parámetro *stride* variable con el objetivo de poder realizar reducciones (*downsampling*) de las imágenes. A su vez, el **bloque del decoder** es bastante similar que el bloque del encoder, a diferencia de que las capas convolucionales (*Conv2D*) se sustituyen por capas de convoluciones traspuestas (*Conv2DTrans*) con tal de poder realizar el aumentado de las imagenes (*upsampling*). Finalmente, el **bloque de salida**, es igual que el bloque de salida del generador 1, detallado en 5.1.2.

Los bloques del encoder y decoder se unen en grupos de $n_{bloques}$. Los bloques de un mismo

Bloque Encoder	
Capas	Parámetros
Conv2D	$kernel$ $filtros$ $stride$ $padding = igual$
BatchNorm	
LeakyReLU	$\alpha = 0.1$

Bloque Decoder	
Capas	Parámetros
Conv2DTransp	$kernel$ $filtros$ $stride$ $padding = igual$
BatchNorm	
Dropout	$razón = 50\%$
LeakyReLU	$\alpha = 0.1$

Bloque de Salida	
Capas	Parámetros
Conv2D	$kernel = 4 \times 4$ $filtros = 3$ $stride = 1$ $padding = igual$
tanh	

Tabla 5.5: Definición de bloques del generador 2.

grupo tienen los mismo parámetros, a excepción del *stride*, que adquirirá un valor de 2 en el último bloque del grupo y de 1 en el resto, de manera que la reducción o el aumento solo se produzca 1 vez por bloque y siempre en la salida.

El generador 2 cuenta con una cantidad variable de grupos del encoder y el decoder. Esta cantidad se define con el hiperparámetro *depth*. Es decir, tanto el encoder como el decoder dispondrán de *depth* grupos. Cada grupo i del encoder, se conectará con el bloque $depth-i$ del decoder. Esto es, el último grupo del decoder recibirá como entrada además de la salida del grupo anterior, la salida del primer grupo del encoder, la del penúltimo recibirá la del segundo y así sucesivamente. La cantidad de filtros de cada grupo del encoder se duplica en cada uno. En el caso del decoder se divide por 2. Cada grupo del encoder i tendrá $filtros = f_{iniciales} * 2^i$, mientras que los del decorer $filtros = f_{iniciales} * 2^{(depth-i)}$. Finalmente, después del decoder se encuentra la capa de salida, que como en el caso del generador 1, se encarga de convertir la salida de la red a una imagen.

5.2 Métricas de evaluación

Uno de los métodos más comunes para determinar la convergencia de las redes neuronales es la estabilización de su pérdida. Como el funcionamiento de las GAN se basa en la *competencia* del generador y el discriminador, cuando la pérdida del generador baja, la

Generador 2			
Bloque	Conexión	Parámetros	Tamaño salida
encoder_1 [e_1]		$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^0$	32×32
encoder_2 [e_2]		$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^1$	16×16
encoder_3 [e_3]		$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^2$	8×8
encoder_4 [e_4]		$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^3$	4×4
		...	
encoder_n [e_n]		$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^{n-1}$	
decoder_n [d_n]	e_n	$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^{n-1}$	
		...	
decoder_4 [d_4]	e_4	$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^3$	4×4
decoder_3 [d_3]	e_3	$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^2$	8×8
decoder_2 [d_2]	e_2	$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^1$	16×16
decoder_1 [d_1]	e_1	$kernel$ $n\_bloques$ $filtros = f_{iniciales} * 2^0$	32×32
salida		64×64	

Tabla 5.6: Definición de la arquitectura del generador 1.

del discriminador tiende a subir y viceversa. Como podemos apreciar en la figura 5.1 hay grandes variaciones de las pérdidas tanto del generador como del discriminador durante el entrenamiento y tampoco permite mucha interpretación de la gráfica, cosa que supone un gran problema para determinar la convergencia del entrenamiento de las GAN.

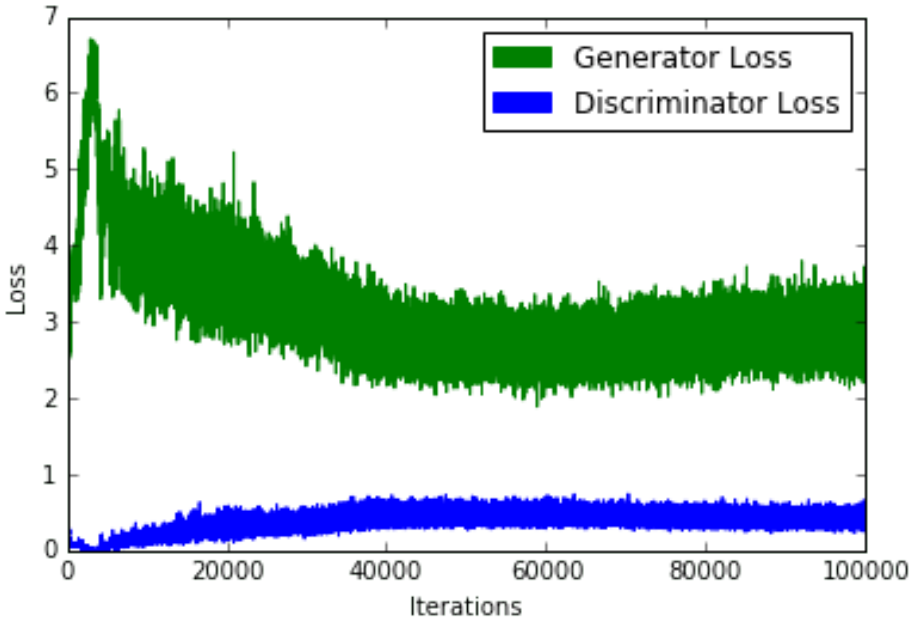

Figura 5.1: Ejemplo de pérdida del generador y el discriminador durante el entrenamiento. Imágen obtenida de https://medium.com/@sanketgujar95/gans-in-tensorflow-261649d4f18d

A consecuencia de la imposibilidad de determinación de la convergencia, esta se ha determinado de manera empírica a partir de un análisis visual de las imágenes durante el proceso de entrenamiento.

Como tampoco se dispone de un método de evaluación de los resultados ni ninguna métrica que los refleje, en los experimentos expuestos a continuación se presentaran ejemplos visuales de los resultados de cada uno de ellos.

Por otra parte, durante los experimentos se ha monitorizado la pérdida del generador y del discriminador con el objetivo de detectar problemas de estabilidad de la GAN. Es común en las GAN que alguna de las redes aprenda más que la otra y acabe teniendo una pérdida de valor 0. En estos casos se produce un colapso y el entrenamiento se estanca sin la posibilidad de mejorar. La estabilidad de las GAN se encuentra cuando las pérdidas del generador y el discriminador se encuentran dentro de unos rangos aproximados y no se producen cambios muy bruscos en las pérdidas. Podemos ver este efecto de estabilidad en la figura 5.1 donde aunque veamos las pérdidas aumenten y decrementen continuamente, son estables durante el tiempo.

5.3 Diseño de experimentos

Los experimentos a realizar se han diseñado en base a la modificación de los distintos huperparámetros de las arquitecturas del generador y la del discriminador. En la tabla 5.7 tenemos un resumen de estos hiperparámetros según lo definido en la sección 5.1.

Discriminador	Generador 1	Generador 2
$f_{iniciales}$	$f_{iniciales}$	$f_{iniciales}$
$kernel$	$kernel$	$kernel$
	$depth$	$depth$
		$n_{bloques}$

Tabla 5.7: Resumen de hiperparámetros de las redes.

A partir de estos, se han definido los siguientes experimentos:

Experimento	Generador	Discriminador
1	**generador 1** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $depth = 3$	$kernel = 3 \times 3$ $f_{iniciales} = 32$
2	**generador 1** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $depth = 5$	$kernel = 3 \times 3$ $f_{iniciales} = 32$
3	**generador 2** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $n_{bloques} = 1$ $depth = 6$	$kernel = 3 \times 3$ $f_{iniciales} = 32$
4	**generador 2** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $n_{bloques} = 2$ $depth = 6$	$kernel = 3 \times 3$ $f_{iniciales} = 32$

Tabla 5.8: Definición de los experimentos.

Además, los experimentos se han entrenado con los dos tipos distintos de entrenamientos (ver la sección 4.1.2, tanto con como sin el módulo de reconstrucción.

5.4 Resultados

En este capítulo se presentan los resultados obtenidos en los distintos entrenamientos que se han llevado a cabo y se realizará un análisis de los resultados obtenidos. Para cada experimento se van a presentar los resultados de ambos entrenamientos en distintos puntos del mismo. En cada columna se presentan los resultados obtenidos para un boceto, presentado arriba, para los dos tipos de entrenamiento, el módulo de reconstrucción (C.R., a la derecha) y sin él (S.R., a la izquierda).

5.4.1 Experimento 1

Experimento	Generador	Discriminador
1	**generador 1** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $depth = 3$	$kernel = 3 \times 3$ $f_{iniciales} = 32$

Tabla 5.9: Definición del experimento 1.

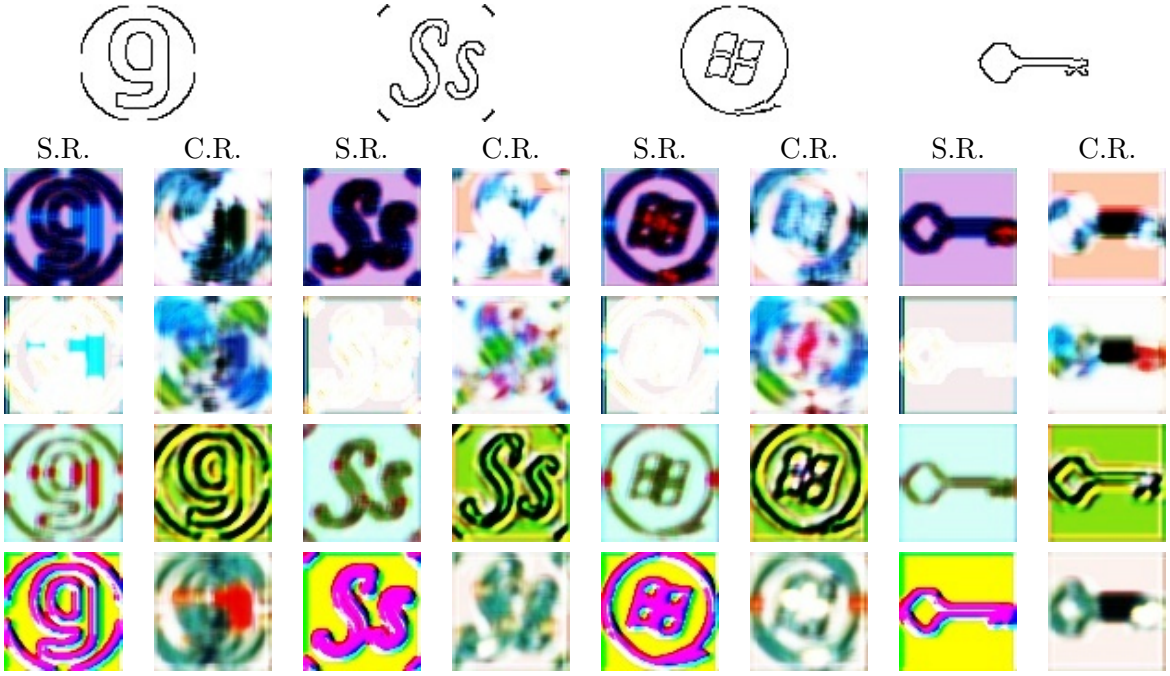

Tabla 5.10: Resultados del experimento 1.

En la tabla 5.10 podemos ver los resultados del **experimento 1**. Se han cogido 4 imágenes al azar del conjuto de *testing* y se ha representado la salida en distintos puntos del entrenamiento. De arriba a abajo, están representadas las salidas del generador para el boceto

dado en la parte superior desde las épocas más tempranas a las últimas. Además, para cada boceto se muestra la salida de la versión del experimento sin el módulo de reconstrucción (a la izquierda) y con el módulo de reconstrucción (a la derecha). Si nos fijamos en los resultados podemos ver que tanto en las épocas tempranas como en las últimas, en ambas variantes del experimento se muestra de manera clara el boceto original, sin perderse apenas detalle. Por otra parte, en ninguno de los dos entrenamientos se obtienen imágenes similares a las del entrenamiento. Las imágenes resultantes, simplemente se colorean de manera homogénea en todas las imágenes en cada época alterando levemente la imagen de entrada.

5.4.2 Experimento 2

Experimento	Generador	Discriminador
2	**generador 1** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $depth = 5$	$kernel = 3 \times 3$ $f_{iniciales} = 32$

Tabla 5.11: Definición del experimento 2.

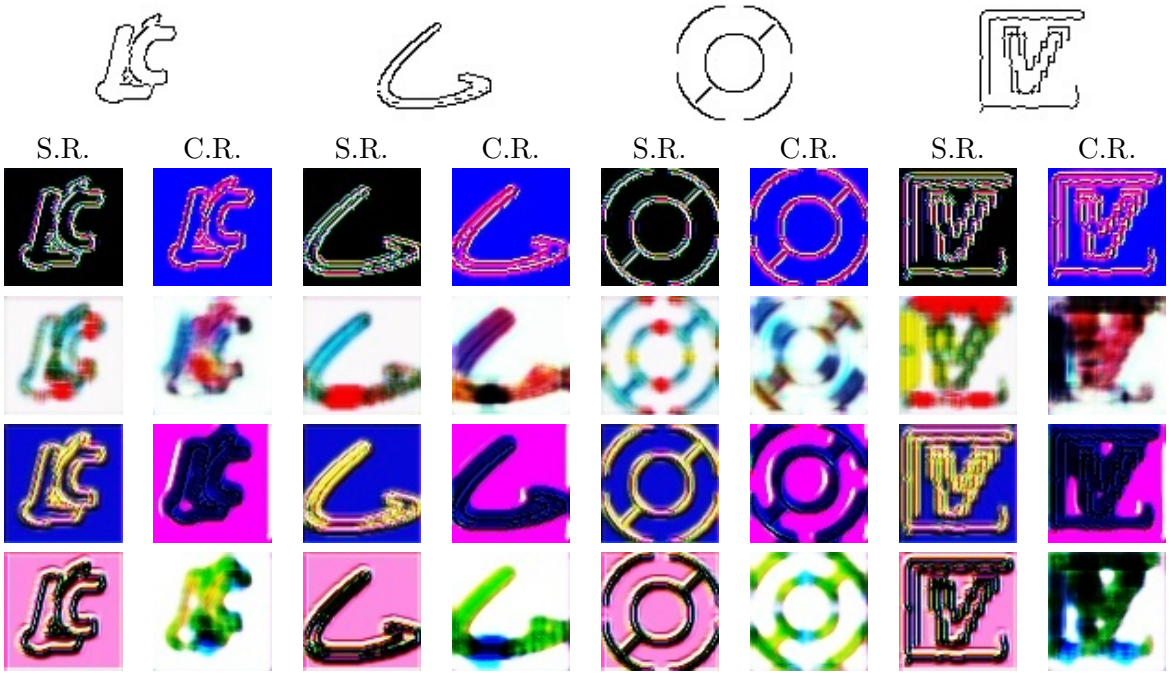

Tabla 5.12: Resultados del experimento 2.

En el **experimento 2**, como podemos ver en la tabla 5.12, los resultados son bastante similares a los del experimento 1. En este caso se puede ver que las imágenes generadas

tienen algo más de complejidad, incorporando relieves o sombras en algunos casos, pero por otra parte, siguen siendo simples y muy similares unos a otros dentro de una misma época de lo que se intuye que los modelos no tienen capacidad de realizar modificaciones sofisticadas sobre los bocetos de entrada. Si nos fijamos en las diferencias entre los modelos entrenados con el módulo de reconstrucción y sin él, no se percibe con claridad cual de ellos respeta más el boceto original. En ninguno de los dos casos se corrompe la entrada del generador y se conservan los detalles.

5.4.3 Experimento 3

Experimento	Generador	Discriminador
3	**generador 2** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $n_{bloques} = 1$ $depth = 6$	$kernel = 3 \times 3$ $f_{iniciales} = 32$

Tabla 5.13: Definición del experimento 3.

Tabla 5.14: Resultados del experimento 3.

En el **experimento 3** vemos un salto cualitativo con respecto a los 2 experimentos anteriores. Como podemos ver en la tabla 5.14, en este caso las imágenes que se generan son de

mayor calidad incluso en épocas muy tempranas. En el caso del entrenamiento sin el módulo de reconstrucción, podemos apreciar como, pese que a en las primeras épocas la imagen de salida no representa con mucho detalle el boceto original, durante el entrenamiento este detalle va creciendo hasta ser más fiel al dibujo. Por otra parte, con el módulo de reconstrucción podemos ver como desde las primeras épocas la imagen resultante no corrompe la original, a excepción del caso de la primera imagen donde los números no se distinguen con claridad. En el resto de imágenes, en cambio, no hay pérdida alguna de detalle. A diferencia de en los anteriores experimentos, podemos apreciar como estos modelos son capaces de contextualizar el boceto y comprender aspectos de más alto nivel. Por ejemplo, reconoce las formas y las colorea de la misma manera o también es capaz de reconocer distintas regiones de la imagen y diferenciarlas en la imagen resultante.

5.4.4 Experimento 4

Experimento	Generador	Discriminador
4	**generador 2** $kernel = 3 \times 3$ $f_{iniciales} = 32$ $n_{bloques} = 2$ $depth = 6$	$kernel = 3 \times 3$ $f_{iniciales} = 32$

Tabla 5.15: Definición del experimento 4.

En los resultados del **experimento 4** que aparecen en la tabla 5.16 podemos ver resultados similares a los de la primera red, no obstante, en las épocas finales, en ambas variantes del entrenamiento, la red tiende a generar imágenes sin sentido o generando patrones aleatorios que parecen no tener relación alguna con la entrada ni parecerse a un logotipo. Si analizásemos las pérdidas del generador y el discriminador, podíamos observar que a mitad del entrenamiento este deja de ser estable y la pérdida del discriminador cae en picado a valores cercanos a 0. En este caso, el discriminador se está sobre entrenando y ha conseguido aprender a discriminar correctamente las imágenes más rápido de lo que el generador es capaz de aprender a generar imágenes. Por ello, el entrenamiento se estanca y no consigue generar mejores resultados. Una posible causa de este problema es que, al aumentar la cantidad de bloques, estamos haciendo que el generador sea demasiado complejo y tenga dificultades para aprender al ritmo del discriminador. Una solución a este problema podría ser disminuir el *learning rate* del discriminador para ralentizar el aprendizaje de este y que el generador pueda aprender al mismo ritmo. Otra posible solución podría ser aumentar la cantidad de filtros del discriminador para que tenga más dificultades de aprender. Por otra parte, este cambio podría ser contraproducente ya que al hacer al discriminador más sofisticado podríamos conseguir el efecto contrario del que buscábamos y hacer que tenga más capacidad de aprendizaje.

5.4.5 Análisis de los resultados

En base a los resultados presentado en esta sección podemos concluir lo siguiente:

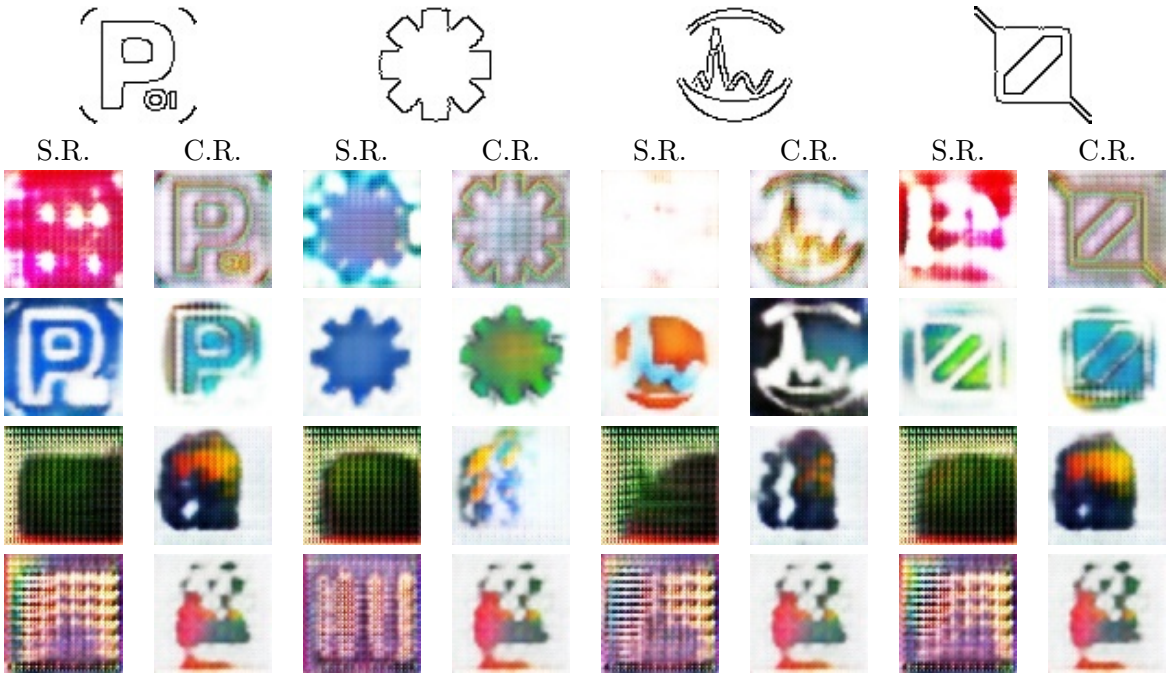

Tabla 5.16: Resultados del experimento 1.

- La realización de cambios locales sobre la imagen (en el caso del generador 1) ayuda a respetar el boceto original. No obstante, los modelos no tienen la capacidad suficiente para realizar modificaciones sofisticadas sobre la imagen y no permite generar resultados de calidad.
- Las arquitecturas encoder-decoder ayudan a mantener el aspecto de la entrada a la vez que proporcionan información contextual y de más alto nivel para la generación de las imágenes consiguiendo resultados más realistas que tengan en cuenta distintas partes del boceto y pudiendo generar además algunos efectos más complejos como sombras o relieves.
- El módulo de reconstrucción ha sido de gran ayuda para el entrenamiento de los modelos. Los experimentos en los que se utilizaba presentaban resultados positivos en épocas más tempranas que los que no lo hacían y consigue que en la mayoría de resultados se mantenga todo el detalle del boceto de entrada.

A continuación se presentan algunos de los mejores resultados del modelo obtenido en el **experimento 3** usando el módulo de reconstrucción de bocetos que era el que presentaba mejores resultados:

Tabla 5.17: Ejemplos de los mejores resultados del modelo del experimento 3 con el módulo de reconstrucción de bocetos. Para cada resultado se muestra el boceto a partir del cual se ha generado y la imagen resultante.

6 Conclusiones

En el presente capítulo, como conclusión del proyecto, se va a hacer una recapitulación del trabajo realizado, así como aportar ideas para futuras líneas de investigación sobre este proyecto.

En este proyecto se ha realizado un sistema de generación de logotipos realistas a partir de un boceto proporcionado como entrada. Esto se ha realizado mediante técnicas punteras en el ámbito de la Inteligencia Artificial como son las tecnologías de GAN. Para el desarrollo del mismo, se ha realizado un estudio previo del estado de la cuestión para conocer sus fundamentos y analizar planteamientos a problemas similares. A partir de estos, se han diseñado las arquitecturas y el entrenamiento de las GAN para, posteriormente, implementarlas con TensorFlow y experimentar sobre ellas. Finalmente, se ha demostrado el funcionamiento del sistema obtenido, que efectivamente genera logotipos realistas cuya base es el boceto proporcionado como entrada.

En cuanto a los resultados obtenidos, se considera que son favorables ya que se ha cumplido el objetivo principal del proyecto, con algunas de las imágenes de ejemplo generando un acabado gráfico muy positivo. Durante la realización del proyecto, por tanto, se han cumplido los siguientes objetivos:

- Se ha realizado un análisis del estado de la cuestión de las GAN mediante la lectura de artículos científicos y se ha realizado una síntesis de las ideas más relevantes.
- Se han estudiado problemas similares y la manera de abordarlos.
- Se ha seleccionado una base de datos adecuada para ser utilizada en el entrenamiento de la GAN, compuesta por imágenes de logotipos reales.
- Se han diseñado, en base al estudio de problemas similares y el análisis del estado de la cuestión, unas GAN capaces de resolver el problema propuesto. Algunas de las arquitecturas, además, introducen un componente original de este proyecto (módulo de reconstrucción de bocetos).
- Se ha realizado una experimentación sobre las GAN diseñadas y se ha realizado un análisis de los resultados.

Por otra parte, el proyecto ha servido para realizar un aprendizaje personal e iniciarme en el mundo de las GAN. Este proyecto me ha permitido estar en contacto con estas técnicas vanguardistas y poder realizar la implementación de una aplicación real que podría abrir las puertas a su utilización en multitud de ámbitos distintos, desde la inspiración de artistas o diseñadores a la hora de crear contenido propio o incluso a la realización totalmente automática de imágenes corporativas utilizándola junto a otras técnicas. Además, debo expresar mi satisfacción personal por la calidad de los resultados obtenidos, a parte también del correcto

funcionamiento del módulo de reconstrucción que se presentaba como una idea original no incluida en otros trabajos previos de este tipo y que ha aportado una mejora en la calidad de los resultados obtenidos. Se puede destacar que la curva de aprendizaje de las GAN y la no experiencia previa en este tipo de tecnología ha reducido el tiempo invertido en la experimentación. Dicho esto, pese que a los objetivos del proyecto se han cumplido, los resultados podrían ser mejores todavía mejores si se consiguieran imágenes más nítidas, sin ruido y con menor pérdida de detalle.

Por último, cabe mencionar que el proyecto abre algunas vías de investigación que pueden servir como punto de partida a otros posibles proyectos futuros. Algunas de ideas de estas posibles vías de continuación son:

- Experimentar con distintas variaciones de las arquitecturas, tanto del generador como del discriminador.
- Evaluar distintas funciones de pérdida que hayan demostrado funcionar correctamente en otras redes GAN.
- Evaluar los modelos obtenidos en un demostrador para comprobar su funcionamiento en casos reales.
- Experimentar con imágenes de mayor tamaño para comprobar el comportamiento de los modelos con un mayor nivel de detalle.
- Desarrollar módulos GAN que potencien la reducción de ruido o el nivel de acabado de las imágenes generadas.

Bibliografía

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-8*(6), 679-698.

Chen, W., y Hays, J. (2018). *Sketchygan: Towards diverse and realistic sketch to image synthesis.*

Clevert, D.-A., Unterthiner, T., y Hochreiter, S. (2015). *Fast and accurate deep network learning by exponential linear units (elus).*

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. En Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, y K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 2672–2680). Curran Associates, Inc. Descargado de `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`

Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., y Seung, H. (2000, 07). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, *405*, 947-51. doi: 10.1038/35016072

Ioffe, S., y Szegedy, C. (2015). *Batch normalization: Accelerating deep network training by reducing internal covariate shift.*

Karras, T., Laine, S., y Aila, T. (2018). A style-based generator architecture for generative adversarial networks. *CoRR*, *abs/1812.04948*. Descargado de `http://arxiv.org/abs/1812.04948`

Kingma, D. P., y Ba, J. (2014). *Adam: A method for stochastic optimization.*

Ledig, C., Theis, L., Huszar, F., Caballero, J., Aitken, A. P., Tejani, A., … Shi, W. (2016). Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, *abs/1609.04802*. Descargado de `http://arxiv.org/abs/1609.04802`

Maas, A. L., Hannun, A. Y., y Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. En *Proc. icml* (Vol. 30, p. 3).

Mirza, M., y Osindero, S. (2014). *Conditional generative adversarial nets.*

Radford, A., Metz, L., y Chintala, S. (2015). *Unsupervised representation learning with deep convolutional generative adversarial networks.*

Ramachandran, P., Zoph, B., y Le, Q. (2017, 10). Swish: a self-gated activation function.

Robbins, H., y Monro, S. (1951, 09). A stochastic approximation method. *Ann. Math. Statist.*, *22*(3), 400–407. Descargado de https://doi.org/10.1214/aoms/1177729586 doi: 10 .1214/aoms/1177729586

Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533–536.

Sage, A., Agustsson, E., Timofte, R., y Gool, L. V. (2017). Logo synthesis and manipulation with clustered generative adversarial networks. *CoRR*, *abs/1712.04407*. Descargado de http://arxiv.org/abs/1712.04407

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(56), 1929-1958. Descargado de http://jmlr.org/papers/v15/srivastava14a.html

Tieleman, T., y Hinton, G. (2012). *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude.* COURSERA: Neural Networks for Machine Learning.

Zhu, J., Park, T., Isola, P., y Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, *abs/1703.10593*. Descargado de http://arxiv.org/abs/1703.10593

Lista de Acrónimos y Abreviaturas

CGAN	Redes generativas antagónicas conditionales o *Conditional Generative Adversarial Networks*.
CNN	Redes neuronales convolucionales o *Convolutional Neural Networks*.
D	Discriminador.
DCGAN	Redes generativas antagónicas profundas convolucionales o *Deep Convolutional Generative Adversarial Networks*.
DL	Aprendizaje profundo o *Deep Learning*.
G	Generador.
GAN	Redes generativas antagónicas o *Generative Adversarial Networks*.
LLD	Large Logo Dataset.
ML	Aprendizaje automático o *Machine Learning*.
NN	Redes neuronales o *Neural Networks*.
RL	Aprendizaje por refuerzo o *Reinforcement Learning*.
SRGAN	*Super Resolution Generative Adversarial Network*.
TFM	Trabajo Final de Máster.